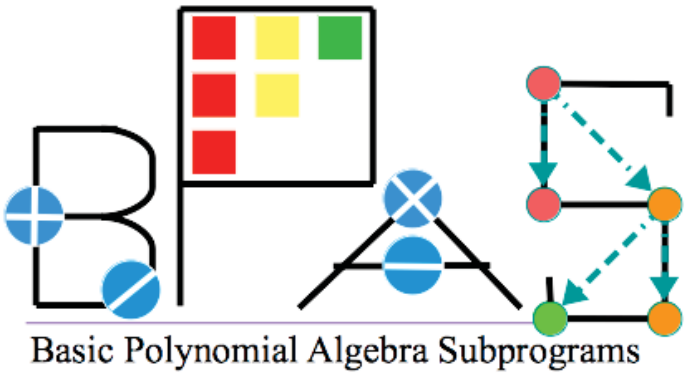

Changbo Chen[1], Svyatoslav Covanov[2,3], Farnam Mansouri[2],
Marc Moreno Maza[2], Ning Xie[2], Yuzhen Xie[2]
[1] Chongqing Key Laboratory of Automated Reasoning and Cognition,
Chongqing Institute of Green and Intelligent Technology, CAS, China;
[2] University of Western Ontario, Canada;
[3] École Polytechnique, Palaiseau, France

Abstract

The Basic Polynomial Algebra Subprograms (BPAS) provides arithmetic operations (multiplication, division, root isolation, etc.) for univariate and multivariate polynomials over prime fields or with integer coefficients. The code is mainly written in `CilkPlus` [11] targeting multicore processors. The current distribution focuses on dense polynomials and the sparse case is work in progress.

A strong emphasis is to put on adaptive algorithms as the library aims at supporting a wide variety of situations in terms of problem sizes and available computing resources. One of the purposes of the BPAS project is to take advantage of hardware accelerators in the development of polynomial systems solvers. The BPAS library is publicly available in source at `www.bpaslib.org`.

1 Design and Specification

Inspired by the Basic Linear Algebra Subprograms (BLAS), BPAS functionalities are organized into three levels. At Level 1, one finds basic arithmetic operations that are specific to a polynomial representation or a coefficient ring, such as multi-dimensional FFTs/TFTs, univariate real root isolation. At Level 2, arithmetic operations are implemented for all types of coefficients rings supported by BPAS (prime fields, ring of integers, field of rational numbers). Level 3 gathers advanced arithmetic operations taking as input a zero-dimensional regular chain, e.g. normal form of a polynomial, multivariate real root isolation.

Level 1 functions are highly optimized in terms of locality and parallelism. In particular, the underlying algorithms are nearly optimal in terms of cache complexity [6]. This is the case for our modular multi-dimensional FFTs/TFTs [14], modular dense polynomial arithmetic [15] and Taylor shift [4] algorithms.

At Level 2, the user can choose between algorithms minimizing work (at the possible expense of decreasing parallelism) and algorithms maximizing parallelism (at the possible expense of increasing work).

For instance, five different integer polynomial multiplication algorithms are available: Schönhage-Strassen, 8-way Toom-Cook, 4-way Toom-Cook, divide-and-conquer plain multiplication and the two-convolution method. The first one has optimal work (i.e. algebraic complexity) but is purely serial due to the difficulties of parallelizing 1D FFTs on multicore processors. The next three algorithms are parallelized but their parallelism is static, that is, independent of the input data size; these algorithms are practically efficient when both the input data size and the number of available cores are small, see [12] for details. The fifth algorithm [3] relies on modular 2D FFTs which are computed by means of the row-column scheme; this algorithm delivers a high scalability and can fully utilize fat computer nodes.

Another example of Level 2 functionality is parallel Taylor shift computation for which four different algorithms are available: the two plain algorithms presented in [4], Algorithm (E) of [8] and an optimized version of Algorithm (F) of [8]. The first two are highly effective when both the input data size and the number of available cores are small. The third algorithm creates parallelism by means of a divide-and-conquer procedure and relies on polynomial multiplication; this approach is effective when 8-way Toom-Cook multiplication is selected. The fourth algorithm reduces a Taylor shift computation to a single polynomial multiplication; this latter approach outperforms the other three as soon as the two-convolution multiplication dominates its counterparts, that is, when both input data size and the number of available cores become large.

This variety of parallel solutions leads, at Level 3, to adaptive algorithms which select appropriate Level 2 functions depending on available resources (number of cores, input data size). An example is parallel real root isolation. Many procedures for this purpose are based on a subdivision scheme. However, on many examples, this scheme exposes limited opportunities for concurrent execution, see [4]. It is, therefore, essential to extract as much as parallelism from the underlying routines, such as Taylor shift computations.

The BPAS library is now being applied to perform symbolic integration of rational functions in [19]. The implementation uses the Lazard-Rioboo-Trager algorithm of [2] to integrate univariate rational functions over the rational numbers. Some subroutines in this code, for instance subresultants [18], are being incorporated to expand the functionality of BPAS. Furthermore, BPAS can compute univariate polynomial GCDs over an arbitrary GCD-domain through the subresultant method.

2 User Interface

Inspired by computer algebra systems like `AXIOM` [10] and `Magma` [1], BPAS makes use of type constructors so as to provide genericity, for instance `SparseUnivariatePolynomial (SUP)` can be instantiated over any BPAS ring. For efficiency consideration, certain polynomial type constructors, like `DistributedDense MultivariateModularPolynomial (DDMMP)`, are only available over finite fields in order to ensure the data encoding a DDMMP polynomial consists only of consecutive memory cells. For the same efficiency consideration, the most frequently used polynomial rings, like `DenseUnivariateIntegerPolynomial (DUZP)` and `DenseUnivariateRationalNumberPolynomial (DUQP)` are primitive types. In other words, `SUP<Integer>` and `DUZP` implement the same functionalities; however the implementation of the latter is further optimized.

Figure 1 shows a subset of BPAS' tree of algebraic data structures. Dark and blue boxes correspond respectively to abstract and concrete classes. BPAS counts many other classes, for instance, `Intervals` and `RegularChains`.

3 Implementation techniques

Modular FFTs are at the core of asymptotically fast algorithms for dense polynomial arithmetic operations. A substantial body of code of the BPAS library is, therefore, devoted to the computation of one-dimensional and multi-dimensional FFTs over finite fields. In the current release, the characteristic of those fields is of machine word size while larger characteristics are work in progress.

The techniques used for the multi-dimensional FFTs are described in [14, 15] while those for one-dimensional FFTs are inspired by the design of the FFTW [5].

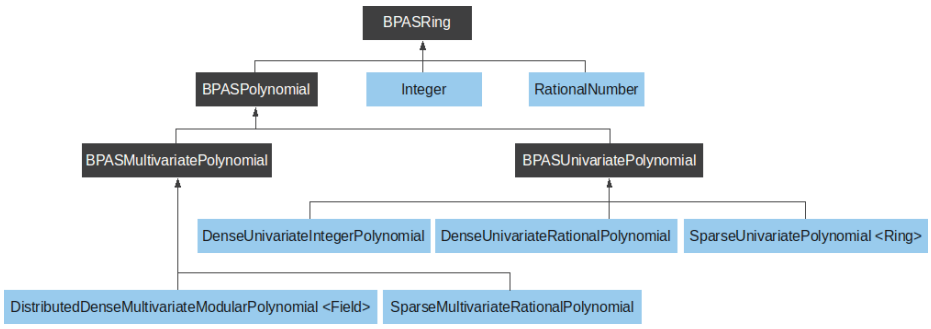

Figure 1: A snapshot of BPAS algebraic data structures.

BPAS one-dimensional FFTs code is optimized in terms of cache complexity and register usage. To achieve this, the FFT of a vector of size n is computed in a divide-and-conquer manner until the vector size is smaller than a threshold, at which point FFTs are computed using a tiling strategy. This threshold can be specified by the user through an environment variable `HTHRESHOLD` or determined automatically when installing the library. At compile time, this threshold is used to generate and optimize the code. For instance, the code of all FFTs of size less or equal to `HTHRESHOLD` are decomposed into blocks (typically performing FFTs on 8 or 16 points) for which straight-line program (SLP) machine code is generated. Instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used (SSE2, SSE4) and instruction pipeline usage is highly optimized. Other environment variables are available for the user to control different parameters in the code generation.

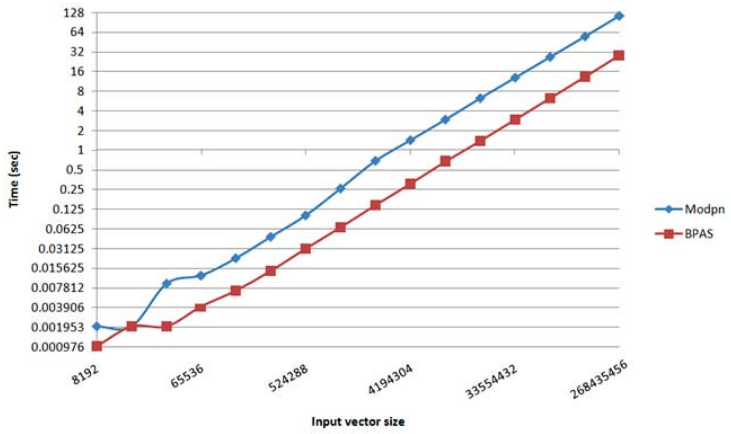

Figure 2: One-dimensional modular FFTs: Modpn vs BPAS.

Figure 2 compares running times (on Intel Xeon 5650) of one-dimensional modular FFTs computed by the Modpn library [17] and BPAS, both using serial C code in this case, where coefficients are in a prime field whose characteristic is a 57-bit prime.

Modular FFTs support the implementation of several algorithms performing dense polynomial arithmetic. As an example, we consider parallel multiplication of dense polynomials with integer coefficients by means of the *two-convolution method* [3]. Given two univariate polynomials $a(y)$, $b(y)$ with integer coefficients, their product $c(y)$ is computed as follows.

(S1) Convert $a(y)$, $b(y)$ to bivariate integer polynomials $A(x,y)$, $B(x,y)$ s.t. $a(y) = A(\beta, y)$ and $b(y) = B(\beta, y)$ hold at $\beta = 2^M$, $K = \deg(A, x) = \deg(B, x)$, where M is essentially the maximum bit size of a coefficient in a and b.

(S2) Consider $C^+(x,y) \equiv A(x,y)\ B(x,y) \bmod \langle x^K + 1\rangle$ and $C^-(x,y) \equiv A(x,y)\ B(x,y) \bmod \langle x^K - 1\rangle$. Compute $C^+(x,y)$ and $C^-(x,y)$ modulo machine-word primes so as to use modular 2D FFTs.

(S3) Consider $C(x,y) = \frac{C^+(x,y)}{2}\,(x^K - 1) + \frac{C^-(x,y)}{2}\,(x^K + 1)$ and evaluate $C(x,y)$ at $x = \beta$, which finally gives $c(y) = a(y)\,b(y)$.

The conversions from the univariate polynomials $a(y)$, $b(y)$ to the bivariate polynomials $A(x,y)$, $B(x,y)$ in Step (S1) as well as the conversions from the bivariate polynomials $C^+(x,y)$ and $C^-(x,y)$ in Step (S3) require only additions and shift operations on machine words. Moreover, the polynomials $C^+(x,y)$ and $C^-(x,y)$ are reconstructed from their modular images (in practice two modular images are sufficient) within Step (S3). Consequently, the data produced by 2D FFT computations is converted in a *single pass* into the final result $c(y)$. Since BPAS' 2D FFT computations are optimal in terms of cache complexity [15], the whole multiplication procedure is optimal for that same complexity measure. Last, but not least, BPAS' 2D FFTs are computed by the row-column scheme which provides lots of parallelism with limited overheads on multicore architectures. As a result, our multiplication code, based on this two-convolution method scales well on multicores as illustrated hereafter.

4 Benchmarks

As mentioned above, one of the main purposes of the BPAS library is to take advantage of hardware accelerators and support the implementation of polynomial system solvers. With this goal, polynomial multiplication plays a central role. Moreover, both sparse and dense representations are important. Indeed, input polynomial systems are often sparse while many algebraic transactions, like substitution, tend to densify data. Parallel sparse polynomial arithmetic has been studied by Gastineau and Laskar in [7] and by Monagan and Pearce in [13]. Up to our knowledge, BPAS is the first publicly available library for parallel dense polynomial arithmetic. For this reason, we compare BPAS' parallel dense polynomial multiplication against state-of-the-art counterpart implementation in FLINT 2.4.3 and Maple 18.

Figure 3 shows the integer case. The input of each test case is a pair of polynomials of degree d where each coefficient has bit size N. The product dN is given by the horizontal axis using a logarithmic scale with radix 2. Timings (in sec.) appear along the vertical axis. Two plots are provided: one for which $d = N$ holds and one for d is much smaller than N. Our experimental results were obtained on an 48-core AMD Opteron 6168, running at 900Mhz with 256 GB of RAM and 512KB of L2 cache.

Acknowledgments

This work was supported by the NSFC (11301524) and the CSTC (cstc2013jjys0002).

References

[1] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.

[2] M. Bronstein. *Symbolic Integration I – Transcendental Functions*. Springer-Verlag, 1997.

[3] C. Chen, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. Parallel multiplication of dense polynomials with integer coefficient. Technical Report, 2014.

[4] C. Chen, M. Moreno Maza, and Y. Xie. Cache complexity and multicore implementation for univariate real root isolation. *J. of Physics: Conference Series*, 341, 2011.

[5] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of IEEE*, 93(2):216–231, 2005.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

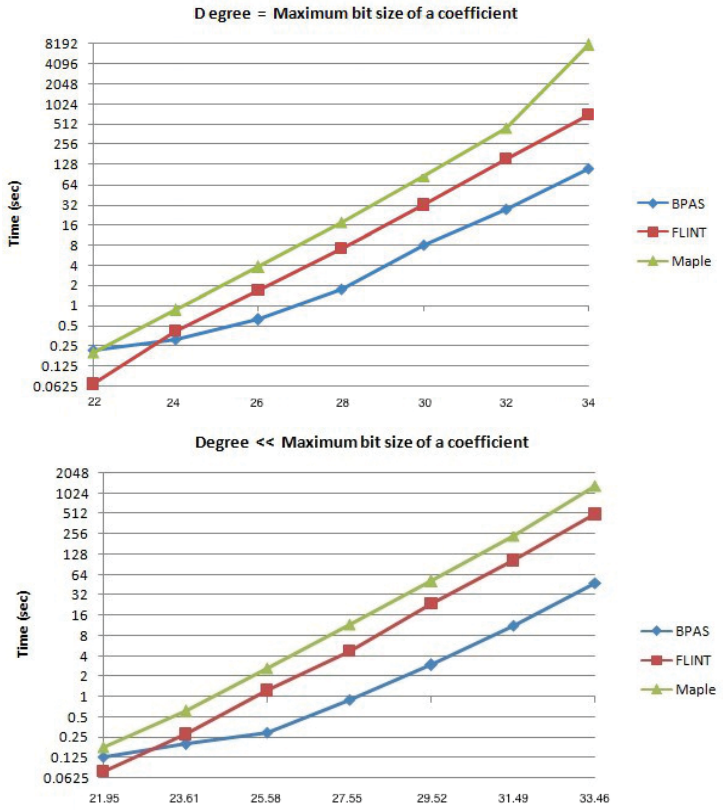

Figure 3: Dense integer polynomial multiplication: BPAS vs FLINT vs Maple.

[7] M. Gastineau and J. Laskar. Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In *CASC*, pages 100–115, 2013.

[8] J. von zur Gathen and J. Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In *Proc. of ISSAC 1997*, pages 40–47, 1997.

[9] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory. Version 2.4.3, `http://flintlib.org`.

[10] R. D. Jenks, R. S. Sutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.

[11] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[12] F. Mansouri. On the parallelization of integer polynomial multiplication. Master's thesis, The University of Western Ontario, London, ON, Canada, 2014. `www.csd.uwo.ca/~moreno/Publications/farnam-thesis.pdf`.

[13] M. B. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *Proceedings of ISSAC 2009*, pages 263–270. ACM, 2009.

[14] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. In *HPCS*, volume 5976 of *Lecture Notes in Computer Science*, pages 378–399. Springer, 2009.

[15] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.

[16] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[17] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. *J. Symb. Comput.*, 46(7):841–858, 2011.

[18] L. Ducos, Optimizations of the subresultant algorithm. *J. of Pure and Applied Algebra*, 145:149-163, 2000.

[19] R. H. C. Moir, R. M. Corless, D. J. Jeffrey. Unwinding paths on the Riemann sphere for continuous integrals of rational functions. *To appear in Proc. of EACA*, 2014.